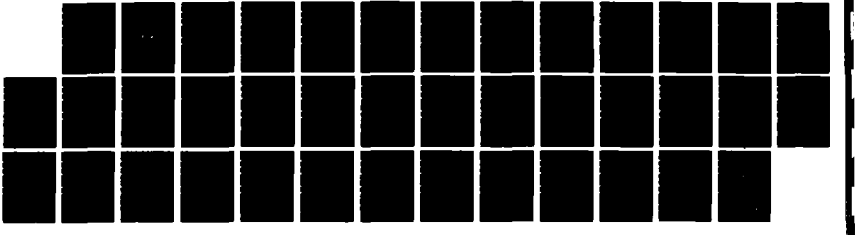
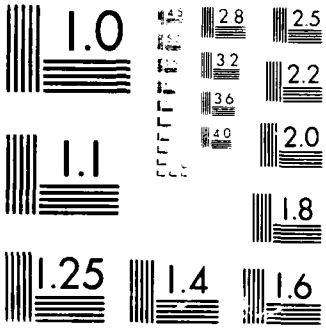
AD-A193 466 COMPARING BARRIER ALGORITHMS(U) COLORADO UNIV AT 1/1
BOULDER COMPUTER SYSTEMS DESIGN GROUP
N S ARENSTORF ET AL JUN 87 CSDG-87-3 NAG-1-648
UNCLASSIFIED F/G 12/7 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

AD-A193 466

DTIC FILE COPY (4)

Comparing Barrier Algorithms

Norbert S. Arenstorf
and
Harry F. Jordan

DTIC
ELECTE
MAR 1 0 1988
S D

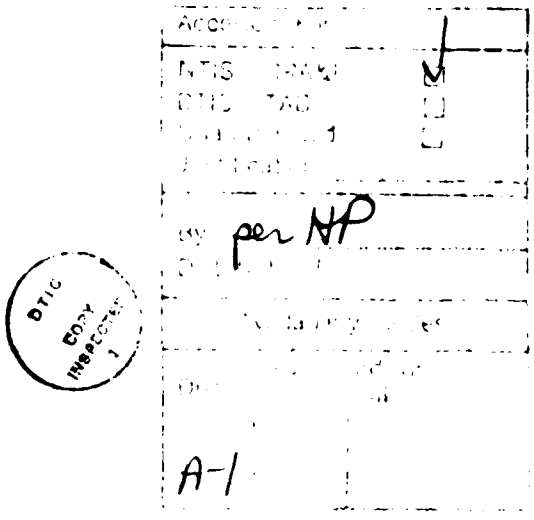
DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

Computer Systems Design Group
Department of Electrical and Computer Engineering
University Of Colorado, Boulder
05/20/87

88 3 1 15 8

Abstract

A barrier is a method for synchronizing a large number of concurrent computer processes. This paper will develop and consider the relative performance of a variety of different barrier algorithms. After considering some basic synchronization mechanisms, a collection of barrier algorithms with either linear or logarithmic depth will be presented. A graphical model is described that profiles the execution of the barriers and other parallel programming constructs. This model shows how the interaction between the barrier algorithms and the work that they synchronize can impact their performance. One result is that logarithmic tree structured barriers show good performance synchronizing fixed length work; while linear self-scheduled barriers show better performance when synchronizing fixed length work with an imbedded critical section. The linear barriers are better able to exploit the process skew associated with critical sections. Timing experiments, performed on an eighteen processor *Flex/32* shared memory multiprocessor, that support these conclusions are detailed.



1. Introduction

A barrier is a method for synchronizing a large number of concurrent computer processes. It is a convenient programming tool if the completion of one part of a parallel program is required before any processes may begin execution of the next part. This paper will develop and consider the relative performance of a variety of different barrier algorithms. The performance of the barrier algorithms will be modeled in terms of a shared memory multiprocessor. Interestingly enough, the interaction of the barrier algorithms with the arrival behavior and departure requirements of the the various processes may impact their performance dramatically. We will see that different barrier implementations can deliver the best performance under differing run time conditions. Actual timing data will be considered.

In an attempt to provide for a fair comparison, all the barrier algorithms will have the following points in common. Each barrier allows for a *sequential code block* to be executed by a single process; that is to say, when executing a barrier all processes will synchronize (the *entry* phase), then one process will execute the sequential part, then all processes will be released (the *signal* phase). All algorithms will reinitialize themselves during use, so that the barriers may be used repetitively in loops, etc. The barrier algorithms should all function correctly, even if one or more of the participating processes are suspended for a finite length of time at any point during execution. Finally, none of the barriers have execution times or data requirements greater than proportional to **np**, where **np** is the number of processes participating in the barrier.

In the process of developing the algorithms, three basic concepts appeared. First, some algorithms require only boolean variables in shared memory, while others use *spinlock* routines that provide for atomic read/write access to shared memory locations. Second, the communication pattern of the entry phase may be either linearly or tree structured. And finally, barriers may have symmetric entry and signal phases, or the signal phase may be implemented as a single broadcast.

2. Argument for correctness of the algorithms

The barrier algorithms will be directed towards a shared memory MIMD multiprocessor. The barriers may require both *shared* variables which can be accessed by all processes, and *private* variables which are unique for each process. For example, data structures used to implement synchronization must be kept in shared memory; **np**, the number of processes, could be stored as either a shared or private variable (since its value should not change); while the process **id**, a unique integer between one and **np** that identifies each process, must be stored in a private location.

2.1. Synchronizing shared variables vs. locks

It should be reiterated that attention will be restricted to synchronization that will work repetitively. Consider three different ways of synchronizing two processes as shown in Figure 2-1. The first algorithm relies only upon shared variables. The state transitions of the shared boolean variables are used as signals between different processes. This algorithm will be contrasted with others that use spinlock routines. These two process synchronization "mechanisms" can be used as building blocks to achieve larger barrier algorithms. In Figure 2-1, the dimensions specified for the data structures are those required by typical algorithms when synchronizing **np** processes.

The first algorithm, (1), works because only two processes access flag(i), and each can cause only one of the two transition/state changes. First, in order to indicate for the master that it has arrived, the slave(i) causes the negative transition of flag(i). Once the master has verified that this negative transition has occurred, then the master responds by sending the slave an exit signal, the positive transition of flag(i). The slave must wait for this positive transition to occur before it is released from the

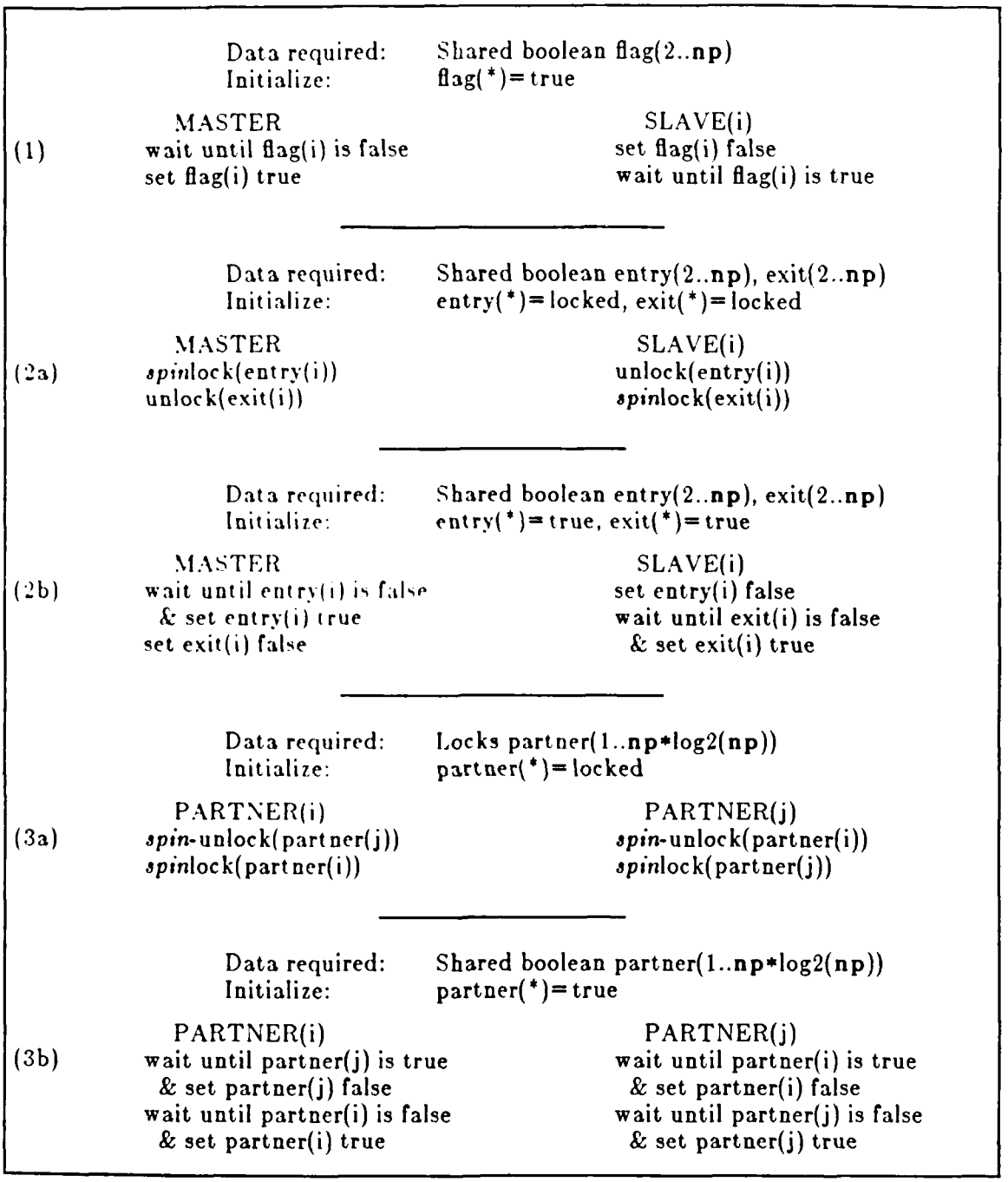


Figure 2-1 "Basic two process synchronization mechanisms"

barrier. There are two wait states, each of which is similar in function to a spinlock. This algorithm splits neatly into entry and signal phases, and the master may execute a sequential code block between its wait and set instructions.

A similar synchronization, (2) can be written using locks.. A *spinlock* will atomically attempt to lock its argument, and it will wait (*spin*) until it is able to do so. Unlock unconditionally clears (unlocks) its argument. Algorithm (2b) is semantically exactly equivalent to (2a), but (2b) expands the spinlock routines into their logical components, allowing for an easier comparison with (1). The slave unlocks a lock that

impedes the progress of the master, and then vice versa. Algorithm (2) is roughly the same as (1), except that the two possible waiting states are implemented with spinlocks, each of which requires a lock; so separate entry and signal data structures must be used, resulting in twice the data requirement of (1). Algorithm (2) does not require the spinlocks to be implemented with atomic read-modify-write cycles.

Another two process synchronization mechanism, one that has been suggested for use with the proposed butterfly barrier [1], is shown in (3). Unlike the other two algorithms, (3) is entirely symmetric with respect to the pair of processes that it synchronizes, which is why the processes are denoted as partners, rather than as master and slave. Again, (3a) is coded using spinlock and unlock routines; (3b) is the semantic equivalent. The butterfly barrier will not be further considered in this paper since its data requirement, **np*log2(np)**, is greater than proportional to **np**, and since it does not allow for single process execution of a sequential code section. The tree barriers, developed below, have a logarithmic depth similar to that of the butterfly barrier, and they require substantially fewer two process synchronization steps than the butterfly barrier.

At first glance, algorithm (3) appears to be superior to (2) because the two processes may proceed in parallel. However, the price to be paid for this is that there is no way to incorporate a sequential code block into this algorithm. More importantly, this algorithm is incorrect for repetitive barriers if it is implemented with an unconditional unlock routine. For example, if PARTNER(i) is suspended after executing its unlock instruction, but PARTNER(j) continues execution and reaches another barrier, then PARTNER(j) may unlock partner(i) a second time before PARTNER(i) has had a chance to lock it. If this occurs, then the barrier fails and some processes will deadlock on the final barrier to be executed. This problem may be corrected if the unlock is replaced with a *spin*-unlock. A spin-unlock would wait until its argument is locked, and then unlock it. With this understanding, we see that (3) has no advantage over (2), since each now has a depth of two waiting statements. Once again, the read-modify-write atomicity of the spinlock routine is unnecessary for this application.

Comparing (1) with (3), one can see that (3) has the same depth as (1), and exactly twice the computation. Algorithm (3) is like a rearranged version of (1) united with its mirror image (master and slave roles reversed), with both halves running in parallel. In this fashion (3) becomes a symmetric version of (1). However, algorithm (1) is simpler and sufficient to synchronize two processes. Only synchronization mechanisms (1) and (2) will be used to develop larger barriers in the sections below.

2.2. Symmetric structure vs. broadcast exit signal

We have seen that the algorithms presented so far have distinct entry and signal phases. The signal phase of a barrier may be implemented as the symmetric image of the entry phase, with the exit signal propagating out in the same fashion as the entry signal was communicated. Or the signal part may be implemented using the broadcast exit/polarity mechanism that will be developed below.

Earlier it was postulated that if a boolean synchronizing variable was used by only two processes, and each could initiate only one of the two possible state changes, then no atomic test and sets, (or other atomic read-write access) would be necessary to insure correct synchronization. The exit phase of a barrier requires a single process to essentially broadcast a signal to all others indicating that they may exit. Perhaps, it is possible for a single separate shared data element to convey the exit information. Only one process should be able to change the state of this exit variable, while all others would have only a read capability.

Some problems come to mind immediately. The exit phase of the barrier may need to serve to *reinitialize* the barrier so that it will function properly on its next iteration. Is it possible to code an algorithm, one that uses a single exit data variable,

that will also correctly reinitialize itself for future iterations? The answer is yes, but it requires an increase in the complexity of the algorithms. The introduction of a private boolean variable indicating the *polarity* of the current barrier iteration is one way to handle the reinitialization problem.

Successive barriers will alternate polarities, as execution of each barrier is encountered in time, dynamically, by all of the processes. All processes will share the same polarity on a given iteration of a barrier, defining the polarity for that barrier iteration. There is no shared variable indicating the overall polarity at a given instant; the polarity is, instead, a private variable for each process, just as the process **id** is private. Note, at a given point in time, one process could be entering a barrier with one polarity, while another process was still exiting the previous barrier of opposite polarity.

A basic broadcast exit signal mechanism is shown in Figure 2-2. The barrier exit signal is the change of state of the shared exit variable. One can see that the slaves will be able to correctly differentiate between successive exit signals, since the following barrier will have the opposite polarity, and all slaves will be inhibited until the the master sends the next exit signal. This exit mechanism provides for a nearly simultaneous release of concurrent processes from a barrier, limited only by how the specific machine architecture handles concurrent reads of a single shared memory location.

	Data requirement:	Private boolean polarity Shared boolean exit
	Initialize:	polarity = true, exit = false
	MASTER	SLAVES
(4)	exit := polarity polarity := not polarity	wait until exit = polarity polarity := not polarity

Figure 2-2 "Broadcast exit signal mechanism"

However this synchronization mechanism does *not* give the master any information as to when and if *all* the slaves have received the exit signal. Any time a signal is sent, there must be a two way flow of information to let the sender know that the signal has been received. If the master issues the next exit signal too soon, before all slave processes have quit waiting on the previous exit state change, then the barrier would be incorrect. This issue is solved if the broadcast exit mechanism is *interleaved* with a correct entry algorithm. In this fashion, the master would issue the exit signal only when assured that all processes have entered the current barrier iteration. Thus, indirectly, the master knows that all processes have "seen" the previous exit signal.

The addition of an alternating polarity to the barrier is compatible with the semantic barrier concept, precisely since *all* processes are required to attend to *each* iteration of a barrier. Thus if all processes are initialized to the same polarity, then we see that it is impossible for processes to get their polarities out of sync, no matter how the barriers are distributed in program code. Thus, a single master can send many slaves an exit signal.

An algorithm somewhat along these lines has been described in [2]. That algorithm uses a system call to signal an *event*; and the operating system insures that all the slave processes waiting for the event to occur, do in fact receive it.

Finally, the algorithms that use synchronization mechanism (1) require the signal phase to reinitialize the data structure for the entry phase. With the use of the

```
                Data requirement:   Private boolean polarity
                                    Shared boolean entry(2..np), exit
                Initialize:         polarity=true, entry(*)=false, exit=false

                  MASTER                          SLAVES(i)
(5)         for each i do
              wait until entry(i)= polarity       entry(i) := polarity
            endfor                                wait until exit=polarity
            exit := polarity                      polarity := not polarity
            polarity := not polarity
```

Figure 2-3 "Linear barrier w/ broadcast exit"

broadcast exit phase, the entry phases will need to reinitialize themselves. It turns out that the same polarity state can be used to modify (1) in order to achieve this end. A resulting linear algorithm, (5), is presented in Figure 2-3. Let the process with **id**=1 be designated the master, while all other processes are slaves. The slaves are indexed from 2 to **np**, with the slave index being equivalent to the process **id**.

In (5) the master receives signals indicating that all slaves have changed the state of their entry variables. Then (and only then) does the master broadcast its change of state. Thus, proper synchronization is insured. A tree structured version of this algorithm with logarithmic depth will be developed in the next section.

2.3. Tree structure vs. linear structure

The next idea to be considered is what type of communication pattern to employ within the entry phase of the barrier algorithm. Either a linear or tree structured approach may be used. A linear approach tends to be simpler since it typically requires fewer overhead calculations. However, a tree structured approach has logarithmic depth. In order to develop a tree structured barrier, two arrangements of the linear algorithm will first be considered. Consider the graphical representation in Figure 2-4 of the same algorithm outlined in Figure 2-3. In Figure 2-4, processes are represented by vertical lines, and time flows downward.

If we think of the basic two process synchronization mechanism in terms of its entry and signal components, then we see that the algorithm in Figure 2-4 works by having a single master accept all the entry signals, then executing the sequential part, and then issuing the exit signal. The ordering of accepting the entry signals is arbitrary, but practical implementations will require a prescheduled, fixed ordering. Usually a Fortran style do loop is employed for this purpose. **Np** boolean shared memory variables are required. This algorithm is coded in an extended Fortran as **bbrlin** (linear broadcast barrier) in Appendix A. Also in the Appendix, a similar barrier, with symmetric entry and exit phases instead of a broadcast exit, is coded as **barlin**.

There is an alternate linear design, as shown in Figure 2-5. Instead of having a single process accept all the entry signals from its slaves; this design has each process accept the entry signal from its next higher numbered neighbor, and then issue its entry signal to the next lower numbered neighbor. In this manner, the entry signals will propagate down to the lowest numbered process. This modified linear algorithm requires a fixed ordering in its communication pattern. The processes numbered 1 and **np** are special cases, thus this algorithm requires additional branching if a single program guides all processes. Thus, this algorithm is less efficient than the previous one.

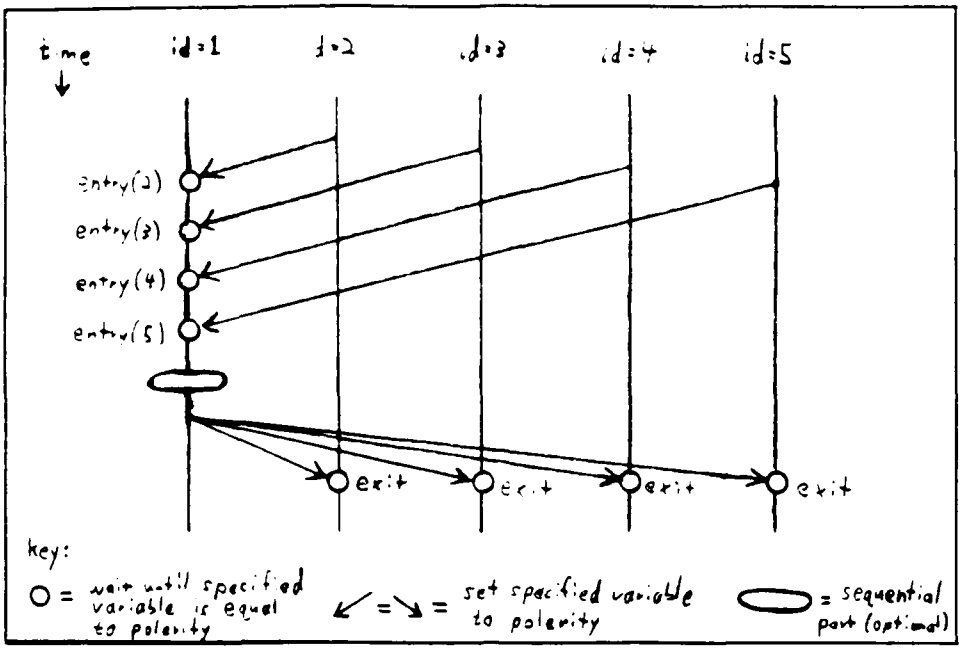


Figure 2-4 "Linear barrier graph (nested entry structure)"

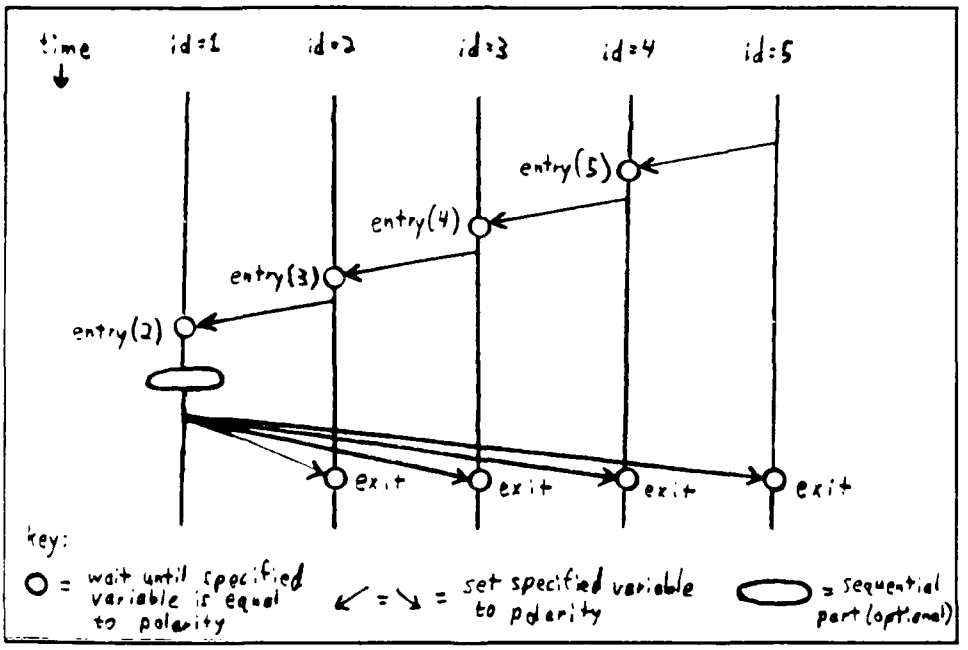


Figure 2-5 "Linear barrier graph (propagated entry structure)"

However, there is a point to be made here. Each of these linear algorithms, using only the two process synchronization mechanisms developed earlier, successfully synchronize many processes. It is possible for a single process to synchronize with several neighbors, and it is also possible for processes to propagate several signals onward to others. The point here is that if one accepts the validity of both of these algorithms, then it is a simple matter to postulate the existence of a binary (or other dimensional)

tree structured algorithm. For the binary tree, at each node a process would accept the entry signal from one neighbor, and then it would propagate this signal along with its own presence to the next lower level of the tree. A graphical version of a tree structured algorithm is presented in Figure 2-6.

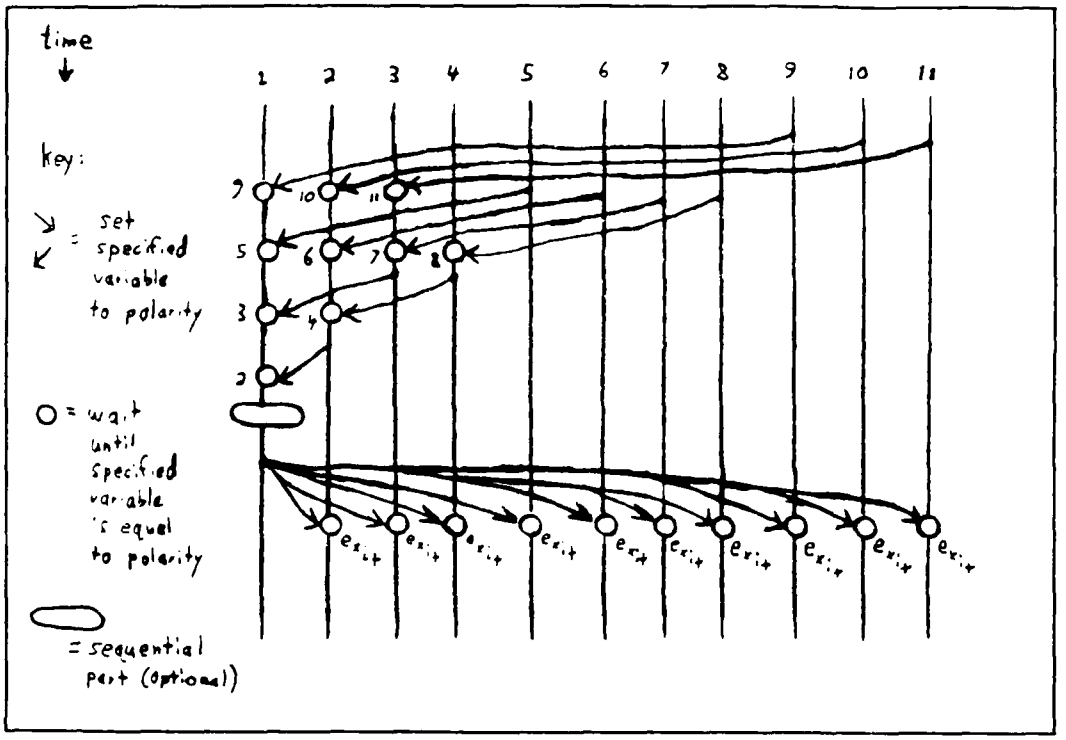


Figure 2-6 "Binary tree barrier w/ broadcast exit"

This is a powerful algorithm. The depth is only 1+roundup(log2(**np**)). As with the linear algorithms, only **np** boolean shared memory data elements are required. This barrier is completely self resetting and airtight, in the sense that if one or more processes are suspended during execution, the barrier is delayed but otherwise continues to function correctly. This algorithm is coded as **bbrtre** (broadcast tree) in Appendix A.

While the depth is logarithmic, each stage of the tree barrier requires more computation, since not only must the synchronization be accomplished, but first each process must calculate with which neighbor to synchronize. This calculation is further complicated if **np** is not constrained to be a power of two. The algorithm shown in Appendix A dynamically calculates with which neighbors to synchronize. It requires about five primitive integer operations (shift, compare, add) in order to calculate the neighbors **id**, at each stage of the entry phase. Surprisingly, a version which precalculates these **id's** and then stores them in a private array, requires nearly as many integer operations to fetch the numbers from the array. However, some run time advantage would probably be achieved using the precalculated approach, at the cost of an additional data structure.

Instead of using the broadcast exit mechanism, it is possible to have a *double* tree structured barrier, with the symmetric entry and signal phases. A graphical version of this barrier is shown in Figure 2-7. For a hard coded example of this algorithm, see **bartre** in Appendix A. The depth is now increased to 2log2(**np**), but we can do away with the polarity concept, simplifying the environment somewhat, and also (**np**-1)

processes are not all competing to read a single exit variable at once. The same shared boolean data structure, an array indexed from 2 to **np**, is used by both the entry and signal parts. In Figures 2-6 and 2-7, the array subscripts are shown in order to indicate which shared variable is being operated on at each point in the tree.

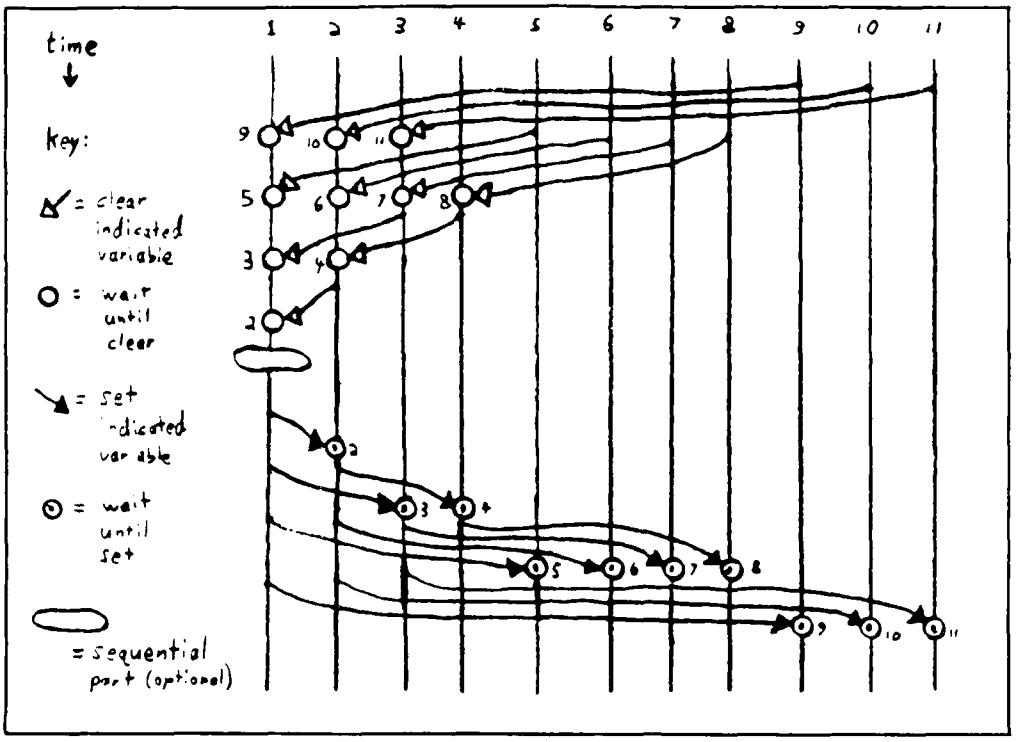


Figure 2-7 "Double tree barrier"

If one wanted to use spinlocks, then it is still possible to employ a tree structured algorithm. Spinlock routines will usually be more expensive, but they may provide superior performance on certain types of hardware (hardware interrupt driven lock tables, for example). Barrier algorithms coded with spinlock routines require separate data structures to be used for the entry and signal parts of the barrier, unless a broadcast signal is used. A tree barrier with broadcast exit and a double tree barrier, both using locks, are coded as **bbrtrl** and **bartrl**, respectively, in Appendix A. These are some barrier algorithms that use two process synchronization mechanisms as their building blocks.

2.4. Linear barriers with critical sections

Another quite different linear approach is possible, one that has traditionally been employed. In this algorithm, a critical section using either an entry or exit lock is used to protect a shared counter variable. Processes count in (except the last), and then spin on the exit lock until the counter is equal to **np**, at which point they count out (except the last). When all processes have counted out, the input lock is reset, allowing the processes to reenter again on the next barrier iteration. Thus, the counter variable swings between one and **np**, and is either (under protection of the entry or exit lock) monotonically increasing or monotonically decreasing, until an endpoint is reached, at which point it is reversed. Careful coding allows the use of only two locks, and each process (except the last) requires two accesses into a critical section per barrier iteration. It should be noted that *fetch and add* [3] hardware could eliminate the

need for a critical section entirely and reduce this algorithm to logarithmic depth.

Processes will be *skewed* in time somewhat as they are go through, one by one, the entry and exit critical regions. The average depth of this algorithm is only **np**, not 2**np**, since the entry and exit phases may *effectively* be overlapped, if there is sufficient work between iterations of the barrier. If there is not enough work to meet this requirement, then the algorithm probably should not be coded in parallel anyway!

This algorithm is shown in Figure 2-8. The pseudo code shown below (Figures 2-8 & 2-9) is to be executed by all processes participating in the barrier. A modified version of this algorithm is coded as **barlok** in Appendix A. Barlok has been modified to allow a sequential part within the barrier, and to insure that the sequential part will always be executed by the same process. On a side note, it may be desirable to insure that the same process always executes the sequential part of a barrier. For example, if there are private variables used within the sequential part on successive barrier iterations, then allowing different processes to execute the sequential parts may introduce unwanted non-determinism into a program's execution. Also, the sequential part of a barrier is often used for file i/o; and on some machine architectures file i/o is simplified if the same process always does the i/o.

```
Data requirement:     Locks ENTRY, EXIT
                      Shared Integer COUNTER
Initialize:           ENTRY=unlocked, EXIT=locked,
                      COUNTER=1

    SPINLOCK(ENTRY)
    If (COUNTER < np) then
      COUNTER := COUNTER + 1
      UNLOCK(ENTRY)
      SPINLOCK(EXIT)
    Endif
    If (COUNTER = 1) then
      UNLOCK(ENTRY)
    Else
      COUNTER := COUNTER - 1
      UNLOCK(EXIT)
    Endif
```

Figure 2-8 "Two lock barrier algorithm"

A version of the two lock barrier that incorporates the broadcast exit/polarity mechanism is given in Figure 2-9. Only one lock for the single critical section is required. Under protection of the critical section, processes decrement the shared counter. The last process to decrement the counter then assumes the role of master and issues the exit signal to all the other processes. Once again, a modified version of this algorithm is coded in Appendix A as **bbrlok**. The modified version allows a sequential part and insures that the process with **id**=1 will always be the one to execute the sequential part.

```
Data requirement:      Private boolean POLARITY
                       Private integer MYCOUNT
                       Shared boolean EXIT
                       Shared Integer COUNTER
                       Locks ENTRY
Initialize:            POLARITY=true, EXIT=false,
                       COUNTER=np, ENTRY=unlocked

        SPINLOCK(ENTRY)
          MYCOUNT := COUNTER - 1
          COUNTER := MYCOUNT
        UNLOCK(ENTRY)
        If (MYCOUNT=0) then
         COUNTER := np
         EXIT := POLARITY
        Else
         wait until EXIT=POLARITY
        Endif
        POLARITY := not POLARITY
```

Figure 2-9 "Single lock barrier w/ broadcast exit"

3. A graphical run time parallel execution model

Why develop so many different barrier algorithms, when they all achieve the same function? Obtaining the best run time execution speed usually is the primary concern. In this section, a graphical model will be used to investigate the run time performance of the barriers. Specifically, the interaction of the barriers with the parallel programming constructs that they synchronize will be examined. The analysis here will not attempt to be exhaustive, it is instead an attempt to gain some insight into the run time behavior of the different barrier algorithms.

Parallel execution within a given programming construct will be modeled using *profiles*. Profiles are shown as two dimensional geometric shapes. A profile includes within its perimeter all the computation corresponding to the programming construct that it represents. On an x,y grid profiles are plotted by processes against time. As in the previous graphs, the processes are plotted along the x axis, and time flows downward along the y axis. Computation internal to a profile is not of interest. What is shown by a profile is the time that each process enters and then exits a given construct. We limit our attention to parallel programming constructs that are executed (on each iteration) by all of the processes.

The power of the model lies in seeing how well different combinations of profiles fit together. This model will consider three categories of parallel programming constructs: parallel work blocks, critical sections, and barriers. This programming model supposes that an arbitrary but fixed number of processes execute a single program consisting of these constructs. The goal is to minimize the execution time of a given sequence of parallel programming constructs. This execution time is modeled by measuring the elapsed distance along the y-axis occupied by the corresponding

sequence of profiles. No portion of a given profile may be superimposed on any part of another profile. If adjacent profiles do not fit together exactly, then the resulting white space is wasted in the sense that processes are just spinning, although this white space may be semantically necessary. A key point here is that other constructs, including the barrier itself, are free to exploit this white space without reducing the overall performance. Timing runs supporting this analysis will be presented in the following section.

Parallel work blocks are assumed to be non-blocking constructs, consisting of some scheduling mechanism which parcels out chunks of single stream work to the various processes. These chunks of work may then be executed in parallel. The area of a parallel work profile corresponds to the total computation and scheduling overhead associated with that parallel work block.

Critical sections provide for mutual exclusion. The profile for a critical section will contain only the computation that a process performs while it is actually within the critical section. The time spent waiting by processes that are temporarily blocked by a critical section is shown as white space in the model. This kind of waiting is caused by the semantic concept of critical section, so it is not appropriate to include it as part of the cost of the implementation of the critical section.

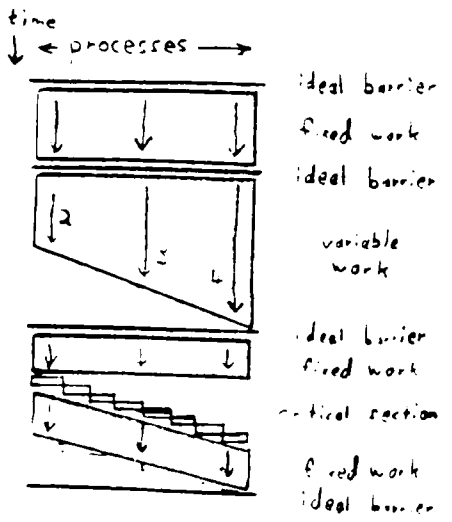


Figure 3-1
"Profiles: ideal barrier"

The barrier, another blocking construct, is treated in a similar fashion. Blocking that is semantically inherent to a barrier will not be included within its profile. Again, this kind of waiting is shown as white space. Consider an *ideal barrier* as shown in Figure 3-1. Since there is no computational overhead associated with an ideal barrier, the profile for this barrier is shown as a horizontal line, with no thickness. It will block the processes that encounter it until all have arrived. If, for example, several processes have encountered a barrier and these processes are waiting for some stragglers, then this waiting is semantically inherent to the barrier and is shown as white space. However, additional waiting or computation required only by the specific implementation of a barrier algorithm will be included within the profile corresponding to that barrier. Thus, the profiles for non-ideal, real barriers attempt to show the overhead costs associated with the barrier. Note, the profile for a correct real barrier must include within its perimeter the profile of the ideal barrier.

The *absolute depth* of a profile refers to the elapsed distance along the y axis, meaning the elapsed time, that a profile would require if it were sandwiched between two ideal barriers. Under certain circumstances, adjacent profiles are able to *overlap* all or part of their execution. Overlap does not refer to physical superposition of the profiles (which is not allowed); instead overlap refers to the situation where some processes are still executing within one profile, while others are already executing within the next profile. Thus, the execution of separate constructs may be partially overlapped in time. If overlap occurs between two or more successive profiles then one can see that the resulting *effective depth* of a sequence of profiles will be less than the sum of the absolute depths.

The degree of overlap between two adjacent constructs depends upon the interaction between the exit and entry contours of the two profiles involved. Entry contours

may be either pre-scheduled or self-scheduled with respect to the process **id**. An *interface* is defined as the interaction of an entry contour with the preceding exit contour. Two general situations can occur. An uneven entry contour that is self-scheduled can overlap as much as possible with the preceding exit contour. Uneven pre-scheduled entry contours may or may not be able to overlap with uneven exit contours, depending on the specific profiles and processes involved. If the interface between successive profiles is pre-scheduled, then we plot the processes along the x axis in the order of their process ids, from one to **np**. However, if an interface is self-scheduled, then the processes are plotted from *fastest* to *slowest*. The fastest process is defined as the first process to enter (or exit) a given iteration of a parallel programming construct. Likewise, the slowest process is defined as the last process to enter (or exit) a parallel programming construct. In this fashion, all the processes may be ranked from fastest to slowest. (Note, the designation of fastest or slowest may vary dynamically among the processes.) The ability to exchange the two orderings, either from one to **np** or from fastest to slowest, requires that the processes involved be fairly homogeneous. These two orderings of the processes will prove useful when analyzing the interfaces between successive profiles.

Three parallel work blocks interspersed with ideal barriers are shown in Figure 3-1. For the sake of simplicity, the optional sequential code blocks of the barriers are ignored. The absolute depth of a parallel work block is given in (6), where **W(id)** is the time that each process requires to do its share of the work. In the case of fixed length work, the **W(id)**'s will all be the same, so the overall depth, $\mathbf{W_{np}}$, is then equivalent to **W(id)**. The absolute depth of a critical section, $\mathbf{C_{np}}$, is given in (7), where **C(id)** is the time each process spends inside the critical section. If **C(id)** is a constant, then $\mathbf{C_{np}}$ simplifies to **np*C(id)**. However, even if each process executes the same code in a given critical section, **C(id)** may not be a strict constant. If the time required for a process to signal that it is exiting a critical section is proportional to the number of processes actively waiting for that signal, then **C(id)** has a dependence on the number of processes seeking access to the critical section.

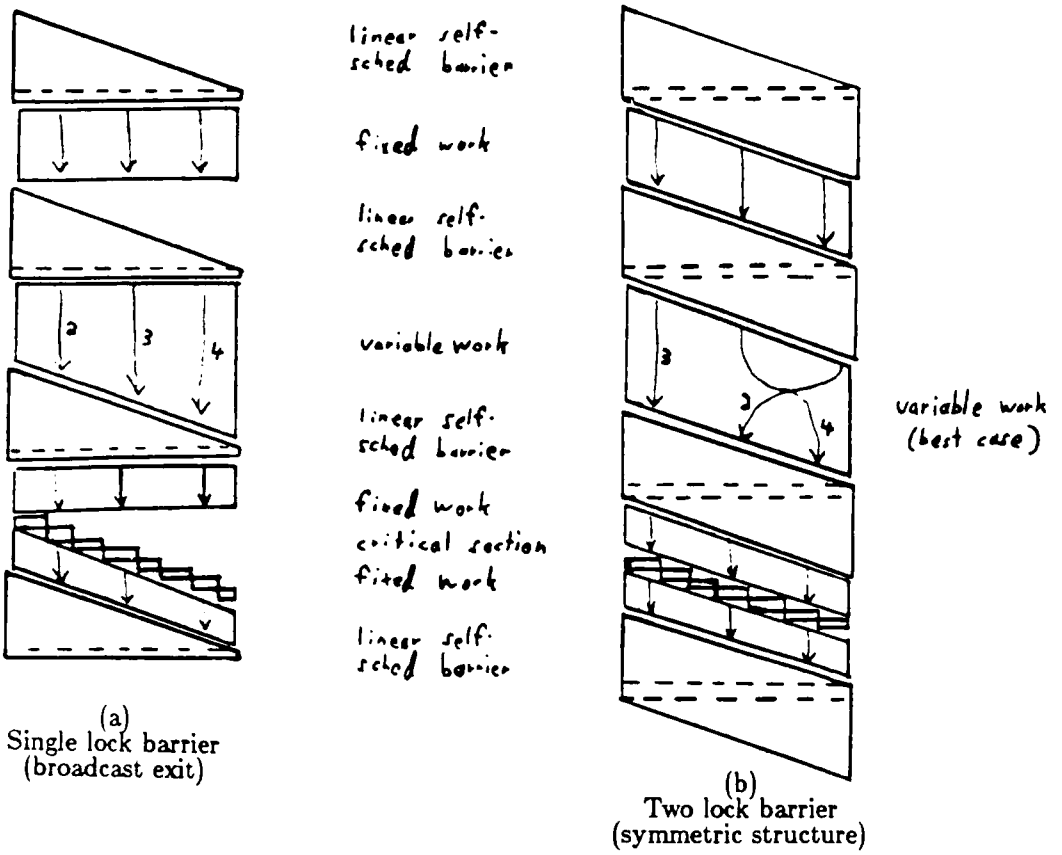
(6) $$\mathbf{W_{np}} = \underset{id=1}{\overset{np}{MAX}}\ \mathbf{W(id)}$$

(7) $$\mathbf{C_{np}} = \sum_{id=1}^{np} \mathbf{C(id)}$$

The eight barrier algorithms developed earlier will be divided into the following classes: linear self-scheduled, linear pre-scheduled, and tree structured. The profiles model will be employed to illustrate some differences in behavior among these classes of algorithms. For each class of barrier, three cases will be examined: barriers interspersed with fixed length work, barriers interspersed with pre-scheduled variable length work, and barriers interspersed with fixed length work with an imbedded critical section. When the barrier and work profiles are combined, the effective depth of the barrier is defined to be the increase in depth over the absolute depth of the work block.

3.1. Linear self-scheduled barriers

The single lock barrier is a linear barrier algorithm with depth proportional to **np**. The profile of the single lock barrier has a self-scheduled entry contour, meaning that the processes may enter in any order, but no faster than one at a time. Self-scheduling is implemented through the use of critical sections internal to the barrier. The two lock barrier is similar to the single lock barrier, except it has symmetric structure, releasing the processes one at a time, as well. Although the absolute depth of the



(a)
Single lock barrier
(broadcast exit)

(b)
Two lock barrier
(symmetric structure)

Figure 3-2
"Profiles: linear self-sched barriers"

two lock barrier is twice that of the single lock barrier, when synchronizing fixed length work, the effective depth of either of these barriers is **np**. This is apparent from Figure 3-2.

If the parallel work is of variable length, then the analysis becomes more complicated. Let the work variation be pre-scheduled (no load balancing employed). Consider, for example, if one work assignment (or process suspension) dominates the work distribution. Considering the single lock barrier, the effective depth of a parallel work block dominated by a single long work assignment will be close to zero. (See Figure 3-2.) As the work load becomes more evenly balanced, the depth of the single lock barrier increases and approaches **np**. For the two lock barrier, if the first process to be released from the barrier receives this long unit of work, this process would be the last one to enter the next barrier iteration, resulting in an effective barrier depth of close to zero. However, if the last process released receives the dominating work assignment, then although this process would still be the last one to enter the next barrier iteration, the effective depth of the barrier is now **np**. Thus, for this example, the average effective depth of the two lock barrier will be **np**/2, still linear but reduced by a factor of two. Other distributions of the variable length work will show similar effects, the degree of the reduction of the effective depth will depend on the exact distribution, however, *on the average*, the effective depth of the two lock barrier synchronizing pre-scheduled, variable length work is between **np** and **np**/2. So, in general, for the case of variable length work, the single lock barrier shows substantial preformance improvement over the two lock barrier. The pre-scheduled variable length work block can be used to approximate the slight variations in the exit contour of a self-scheduled parallel work block.

For the case of fixed length work with an imbedded critical section, we see that both the single lock and two lock barriers are close to ideally efficient! Since the critical section requires processes to arrive in skewed order for maximum efficiency, we see that it does not hurt if the barrier implementation lets the processes out in a skewed fashion. And since the processes leave the critical section in a skewed manner as well, then if the barrier requires a skewed entry, no additional performance penalty is incurred, as shown in Figure 3-2a.

3.2. Linear pre-scheduled barriers

Instead of using critical sections, the pre-scheduled linear algorithms require a single master to accept the entry signals from all other processes, one at a time, in a predetermined order. The pre-scheduled barriers require less time to complete each stage of their algorithms, since they do not require a critical section at each stage. A profiles model of the linear pre-scheduled barrier with broadcast exit is shown in Figure 3-3.

The linear barrier with broadcast exit has a depth of only **np+1** for the case of fixed length work. The symmetric pre-scheduled linear barrier occupies the master process throughout both the entry and exit parts of the barrier. Thus, the effective depth of this linear barrier remains **2*np** for fixed length work, since the master must also perform its share of the work. If the work distribution becomes variable, the pre-scheduled linear barriers also show reduced depth but not to the extent of their self-scheduled counterparts. For example, the worst case scenario pictured in Figure 3-3 could not happen if the entry contour of the barrier was self-scheduled.

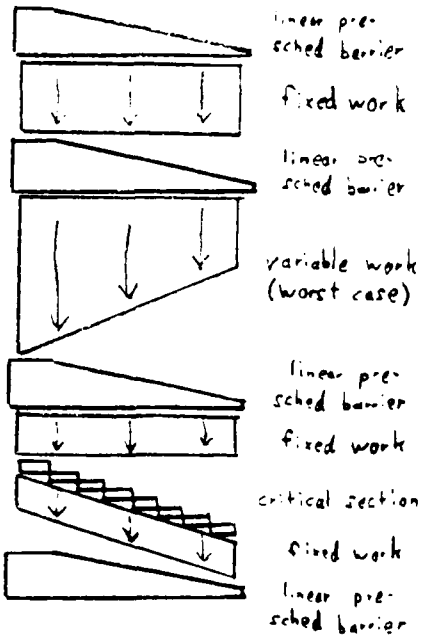


Figure 3-3
"Profiles: linear pre-sched barrier w/ broadcast exit"

When synchronizing critical sections, the effective depth of the pre-scheduled linear barriers is not as good as the effective depth of their self-scheduled counterparts. If the processes go through the critical section in the optimal order, ie, master first, then slaves in the order of their process ids, then the effective depth of the barrier will be close to optimal. However, if the master happens to be the last process to go through the critical section, then the next barrier iteration will have its full depth. Thus, one would expect some reduction of the effective barrier depth when synchronizing work containing critical sections, but not the near optimal behavior of the self-scheduled linear barriers.

3.3. Logarithmic (tree) barriers

The tree barriers have depth logarithmically proportional to **np**. The profiles for the tree barriers, as shown in Figure 3-4, are quite simple, since they are rectangular in shape, with flat entry and exit contours. The tree barriers also are analyzed for each of the three conditions above, fixed length work, variable length work, and work containing a critical section, however, the analysis will be much simpler. Unlike the linear barriers, the effective depth of the tree barriers is nearly independent of the type of parallel work that they synchronize. No matter in what order processes arrive, each process, including the last, must go through all the stages of the tree. If the process

arrival times are skewed, some variance in the effective depth results since the time a process spends at each stage varies slightly depending on whether it is playing a master or slave role at that node. But this is only a minor effect. Thus, even if there is a wide variance in process arrival times, the effective depth of the tree barrier remains nearly constant.

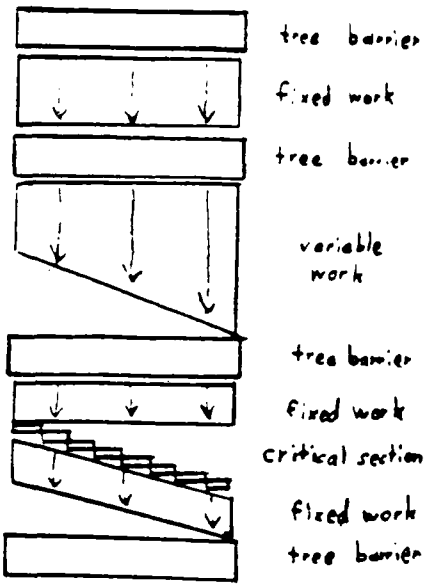


Figure 3-4
"Profiles: tree struct. barrier"

The analysis for tree barriers with broadcast exit is similar. The only difference is that the broadcast exit reduces the depth from 2*log2(np) to log2(np)+1. If locks are used, then each stage of the tree would be expected to have a longer execution time compared to trees that use only boolean variables, resulting in a longer total execution time. Otherwise, the analysis is unchanged.

4. Timing results

Timing runs on an actual shared memory multiprocessor support the predictions derived from the *profiles* parallel execution model developed above. Several experiments timing all eight barrier algorithms were run on a Flexible Computer Corporation *Flex/32*. In order to evaluate the effect of the number of processes (np) on barrier performance, np was varied from two to eighteen in increments of two. Barriers synchronizing fixed length work, variable length work, and fixed length work with an imbedded critical section were timed in three separate experiments.

4.1. Methodology

Each experiment consisted of nine trials; one trial for each of the eight barrier algorithms, and one trial simulating the behavior of the *ideal barrier*. For each of the eight "regular" trials, a barrier followed by the parallel work block corresponding to that experiment was placed in a loop, and execution of this loop was timed for 100000 iterations. The timing of the ideal barrier when synchronizing the various work blocks was simulated using some algorithms described below. The elapsed time of the ideal barrier trial was then subtracted from the elapsed times for each of the other eight trials. These resulting times were then divided through by the number of iterations of the loop, yielding a measure of the per iteration overhead (effective depth) imposed by each of the eight barrier algorithms.

The first experiment consisted of timing barriers that synchronized a fixed length parallel work block. The fixed length work required each process to execute 30 iterations of single precision multiply additions (mul-adds) and some associated subroutine linkage. This number of mul-adds is sufficient to insure that successive barriers will not attempt to overlap with each other. Strictly private operands were used. A barrier followed by the fixed length work block was placed in a loop and timed for 100000 iterations. This measurement was repeated for each of the eight barrier algorithms. The ideal barrier timing loop was simulated using a very simple algorithm: time only the fixed length work for 100000 iterations.

The second experiment timed barriers synchronizing variable length, pre-scheduled, parallel work blocks. In a set up phase, processes iteratively filled private arrays, called **myrand**, with 100000 random numbers. Processes also cooperated to determine the maximum random number that was generated on each iteration, and these maximum values were stored in separate private arrays, called **maxval**. The random numbers ranged between 30 and 59, with a flat distribution among these values. A linear congruential random number generator was used, and each process calculated an initial private seed by adding its process **id** to a single shared "starter" integer. Each iteration of the variable length work block required the processes to pull a random number from their **myrand** arrays, and then they would execute that many iterations of single precision mul-adds. For the timed part of this experiment, a barrier followed by the variable length work block was placed in a loop and timed for 100000 iterations. The ideal barrier timing loop was simulated by timing a single process executing 100000 iterations of "work" with no barriers; each "work" iteration consisted of retrieving the maximum random number from the **maxval** array and then executing that many mul-adds. In this manner an ideal barrier is simulated, since an ideal barrier would have to wait, on each iteration, for the process with the most work to finish.

Finally, the third experiment timed barriers synchronizing fixed length work with an imbedded critical section. Each of these work blocks consisted of 15 mul-adds, followed by a critical section, which enclosed a single mul-add, followed by 15 more mul-adds. Since each of the **np** processes must execute the critical section in turn along with its private mul-adds; the ideal barrier timing loop is simulated by timing 30+**np** mul-adds and **np** subroutine linkages (in order to simulate the effects of the spinlocks) on each iteration. This experiment suffers from the difficulty of approximating the ideal barrier exactly, since the bus contention produced by the critical section cannot be exactly accounted for, and this overhead could thus be incorrectly attributed to the barriers.

These three experiments are interesting because they approximate some typical parallel work scheduling mechanisms: pre-scheduling and self-scheduling [2]. Pre-scheduling does not have the synchronization overhead required by self-scheduling and is efficient when work iterations are constant in their execution time. Pre-scheduling is often employed to schedule (non-branching) parallel loops. With pre-scheduling, work iterations are divided up evenly among processes, irrespective of the execution time required by each iteration. As is shown in Figure 4-1, if processes enter a homogeneous pre-scheduled work block in unison, they probably will exit in a step function, since **np** likely will not divide evenly into the number of work iterations. The first experiment timing fixed length work approximates a pre-scheduled parallel loop where **np** does divide evenly into the number of loop iterations. However, even if processes exit in a step function, we still have the situation where many processes exit the work block at once.

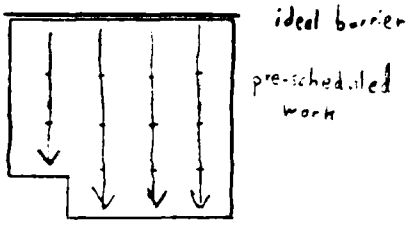
Self-scheduling provides for load balancing and is efficient when work iterations vary in their execution times. With self-scheduling, under protection of a critical section, processes take the "next" available work descriptor from a shared scheduling mechanism whenever they are ready for additional work. In spite of the load balancing concept, processes will be somewhat skewed in time as they exit a self-scheduled parallel work block. This skew results from variance in work execution times and/or the effect of the critical section used to schedule the work iterations. The second and third experiments approximate barriers synchronizing self-scheduled work since they model variable length work and the effects of critical sections, respectively.

4.2. Computing environment

These three experiments were run on a Flex/32 shared memory multiprocessor consisting of a shared memory store and a set of single board microcomputers with true private memory on each board. Processes may be bound to processors, so unexpected process suspension is not a major issue. The Flex/32 supports efficient implementation of spinlocks through a hardware test and set machine language instruction that is available to the user. Memory accesses into private data structures and instruction fetches do not interfere with shared memory cycles on the dual common bus. A proprietary architecture interfaces local busses with a dual common bus connected to shared memory. Unfortunately, Flexible Computer Corporation has not published detailed descriptions of these interfaces. Flexible provides MMOS, its distributed "multicomputing" operating system [4], to supervise parallel programs. The Flex/32 used belonged to the NASA/Langley Computational Structural Mechanics Group in Hampton, Virginia. NASA's Flex/32 is configured with eighteen processors able to run in parallel.

4.3. Sources of timing error

Before describing the curves, let us examine some sources of timing errors. The clock function on the Flex/32 is implemented through software using a system interrupt. These system interrupts increment a private clock (an integer variable). The timer granularity was one second; since 100000 iterations were timed, the timing granularity per iteration is is reduced to .01 millisecond (ms). Since the effective depth of the barriers per iteration was calculated as the difference between two elapsed times, the error due timing granularity (per iteration) is within ± .02 ms. Over twenty four hours of parallel cpu time was required in order to achieve this low timing granularity.



Example: Let np = 4. If there are 15 units of work to be scheduled, with pre-scheduling each process receives either 3 or 4 work iterations.

Figure 4-1
"Profiles: a pre-sched work block"

There is an additional source of error. Each processor receives the system interrupt at the specified frequency. These interrupts occur asynchronously for the processes in a round robin fashion. The duration of this interrupt has been measured to be approximately .3 ms in duration. The timing program was compiled with a configuration specifying only one system interrupt per second, hence a one second timing granularity. A .3 ms interrupt per second represents only .03% of cpu time per processor. However, since the interrupts occur asynchronously, when all 18 processors are being used, then during .54% of the time one of the processors will be servicing the interrupt. This is the parameter of interest when timing barriers! Fortunately, this magnitude of process suspension will not significantly distort timing measurements. It should be noted that NASA's Flex/32 has a default configuration of 50 MMOS system interrupts per second. While this configuration reduces the timer granularity to 20 ms, the percentage of time during which one of the processors is suspended is increased to a whopping 27%, clearly unacceptable for timing barriers.

4.4. Results

The results for each of the three experiments are plotted in Figures 4-2, 4-3, and 4-4. Each figure shows curves corresponding to the various barriers, with different values of **np** plotted against *effective* execution time (milliseconds per barrier iteration). The effective execution time of a barrier is defined as the difference between the execution time of the work and barrier combination and the execution time of the work synchronized by an ideal barrier. All three figures are plotted using the same time scale, allowing for comparison between figures.

For the case of barriers synchronizing fixed length work, Figure 4-2 plots the observed effective depth of all eight barriers against **np**. One observation is that the logarithmic tree barriers show better performance than the linear barriers, even for small values of **np**. Also, the broadcast barriers (those using the polarity exit mechanism) show superior performance than their symmetric (identical entry & signal structure) counterparts. Another observation is that the barriers that use spinlock routines show marked performance degradation as **np** becomes large. This effect may be attributed to increased bus competition that forces shared memory bus requests to line up in a queue. If many processes are competing for access to a lock, one might think that no performance degradation would result, since *one* of the processes should be succeeding, even if others are having their bus requests delayed. This is indeed the case; however, inefficiency is introduced when the owner of a lock must compete for shared bus access in order to *unlock* it. If 18 processes are competing randomly, the average unlock command requires around 17 attempts before succeeding, assuming a single shared bus with random arbitration. The situation with Flexible's dual bus is less clear, but this same type of effect is probably occurring. Since the critical sections themselves are very short, increasing the number of bus cycles required by the unlock instruction will significantly degrade performance, and this is evident from the plot. Apparently, it is the read-modify-write cycles that place the greatest burden on the shared memory bus. The bus contention appears to be much less for the barriers that do not use spinlocks.

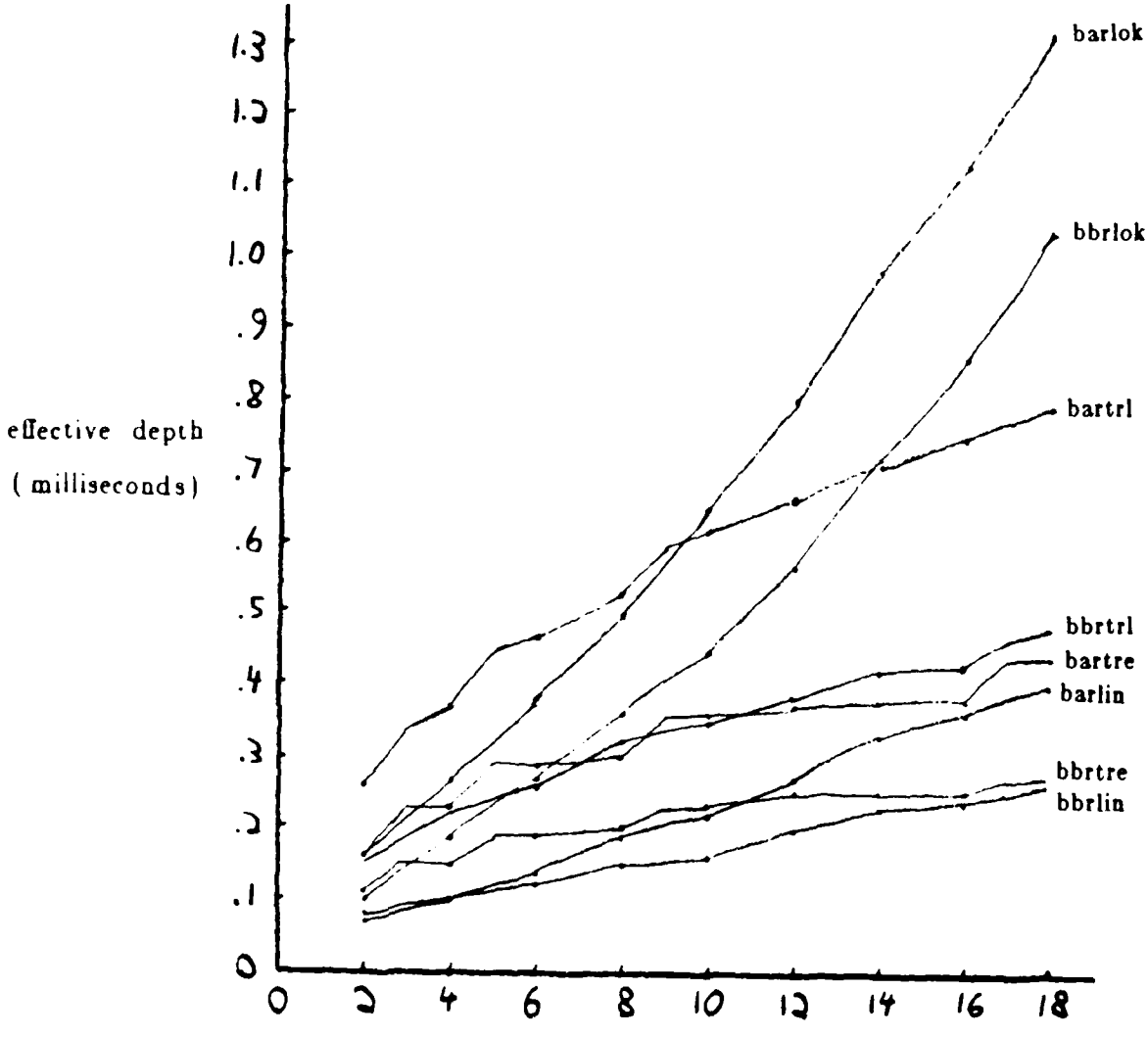
Figure 4-3 shows the barrier performance when synchronizing variable length parallel work blocks. As the profiles model predicts, the linear barriers are better able to exploit the variation in process arrival times. One interesting feature is that the linear barriers with broadcast exit, do a better job than those that have linear exit phases. In efforts to prevent visual clutter, Figures 4-3 and 4-4 do not plot curves for the tree barriers that use locks. However, timing measurements were made for these algorithms as well, and the tree barriers with locks showed substantially greater overhead than the tree barriers without locks.

Figure 4-4 plots the barrier performance when synchronizing work blocks with critical sections. For the case of critical sections, the barriers that had the worst performance in the fixed length work experiment now show the best performance! Even for the larger values of **np**, the linear barriers have a small depth that remains nearly independent of **np**, whereas the tree structured barriers show their normal logarithmic growth. Although Figure 4-4 may look cluttered, it would be misleading to provide more detail by enlarging the time scale since the timing granularity is within ± .02 ms.

5. Conclusions

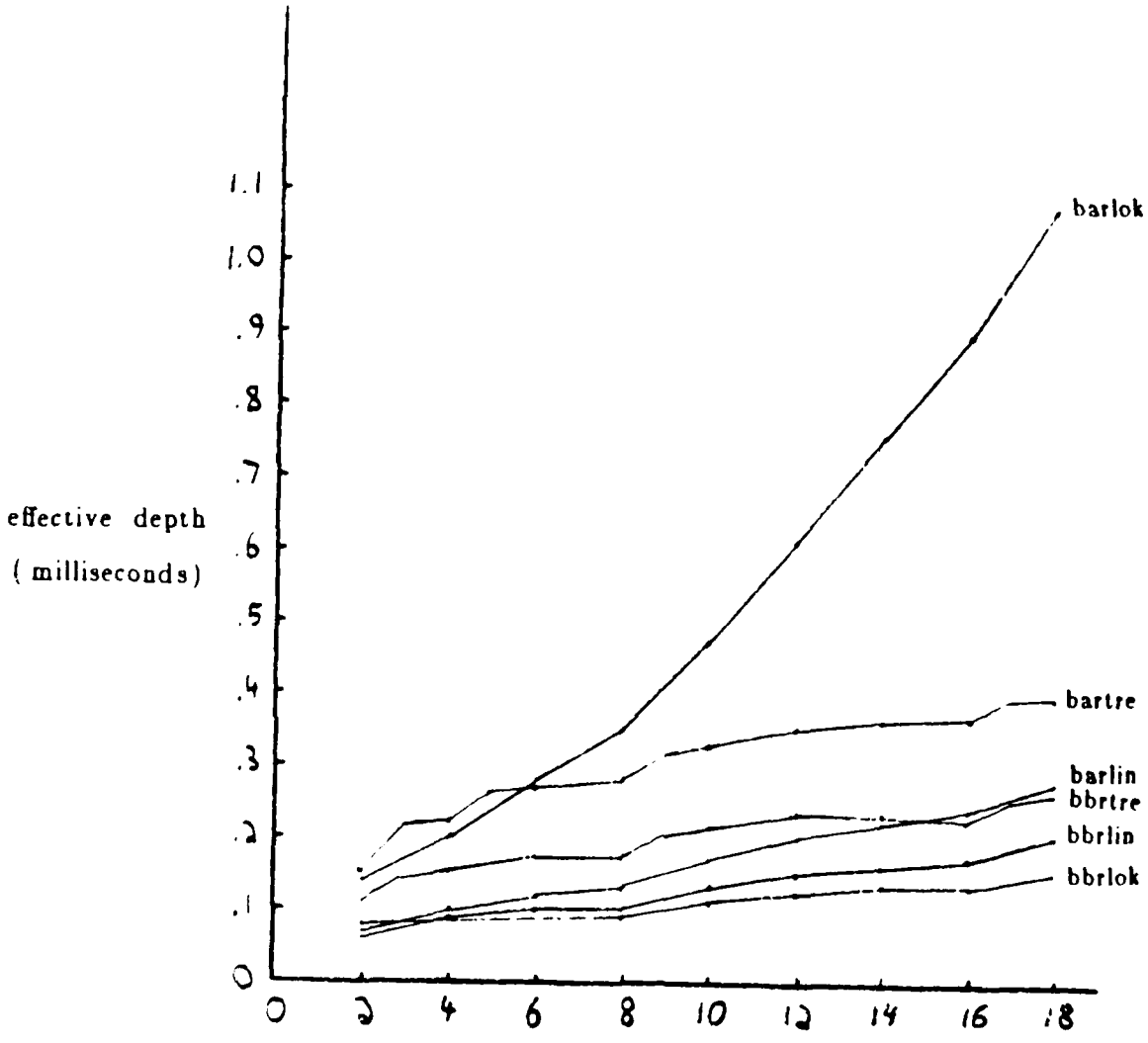
Three concepts were isolated in the development of these barrier algorithms. A barrier may have linear or tree structured communication patterns. A barrier may have symmetric entry and signal phases, or the signal phase may use a single broadcast exit signal. And synchronization within a barrier may rely solely upon memory accesses into shared data structures, or algorithms may use locks and their associated

Figure 4-2 "Timing results for barriers synchronizing fixed-length work"



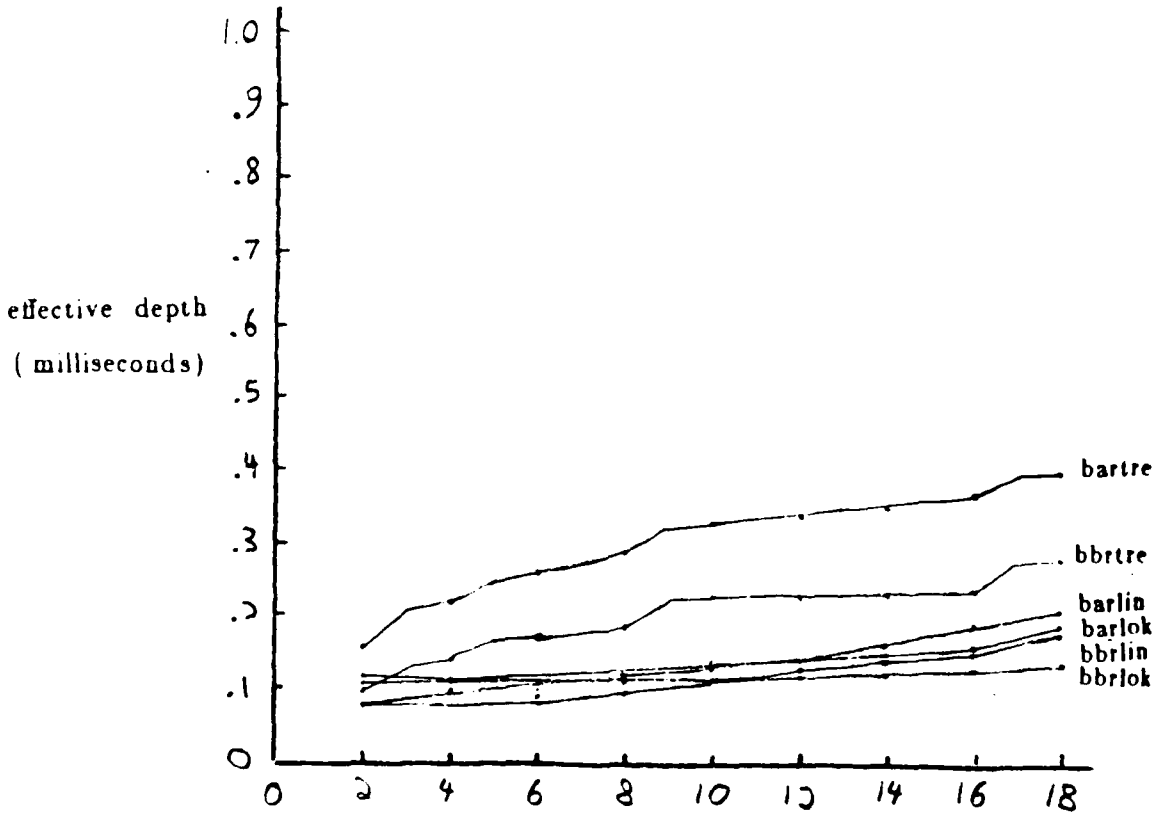
barlin:	linear (pre-sched),	no locks,	symmetric entry & signal phases
barlok:	linear (self-sched),	locks,	symmetric
bartre:	tree structured,	no locks,	symmetric
bartrl:	tree structured,	locks,	symmetric
bbrlin:	linear (pre-sched),	no locks,	broadcast exit signal
bbrlok:	linear (self-sched),	locks,	broadcast
bbrtre:	tree structured,	no locks,	broadcast
bbrtrl:	tree structured,	locks,	broadcast

Figure 4-3 "Timing results for barriers
synchronizing variable-length work"



barlin:	linear (pre-sched),	no locks,	symmetric entry & signal phases
barlok:	linear (self-sched),	locks,	symmetric
bartre:	tree structured,	no locks,	symmetric
bbrlin:	linear (pre-sched),	no locks,	broadcast exit signal
bbrlok:	linear (self-sched),	locks,	broadcast
bbrtre:	tree structured,	no locks,	broadcast

Figure 4-4 "Timing results for barriers synchronizing critical sections"



barlin:	linear (pre-sched),	no locks,	symmetric entry & signal phases
barlok:	linear (self-sched),	locks,	symmetric
bartre:	tree structured,	no locks,	symmetric
bbrlin:	linear (pre-sched),	no locks,	broadcast exit signal
bbrlok:	linear (self-sched),	locks,	broadcast
bbrtre:	tree structured,	no locks,	broadcast

spinlock routines. For the sake of completeness, and to provide for a thorough foundation upon which to make comparisons, all eight combinations of these three concepts were realized as barrier algorithms. Specifying which of these barriers is the "best" is not so easy a task, since there are several trade offs involved and different machine architectures may favor different barrier implementations.

5.1. Analysis

If we have parallel routines that can be executed by an arbitrary number of processes, then the *speedup* of a parallel routine can be defined to be the ratio of the single process execution time of the routine (without synchronization overheads) against the parallel execution time (including synchronization overheads). Introducing optimized barriers into existing programs tends to result in only minor improvement in the speedup if these programs were not "barrier bound" to begin with. Optimizing the barrier's execution time delivers instead a different payoff: the threshold size of work blocks that may profitably be parallelized is decreased. Equation (8) shows the formula for speedup when considering only a single parallel work block followed by a barrier, where $\mathbf{W_{np}}$ is the time required for **np** processes to execute the parallel work, $\mathbf{W_1}$ is the single process execution time, and $\mathbf{B_{np}}$ is the effective barrier execution time. If $\mathbf{W_{np}} >> \mathbf{B_{np}}$, then decreasing $\mathbf{B_{np}}$ will not improve the speedup by much. However, if $\mathbf{B_{np}}$ is of similar or greater magnitude than $\mathbf{W_{np}}$, then decreasing $\mathbf{B_{np}}$ will substantially increase the speedup. Thus reducing $\mathbf{B_{np}}$, the barrier effective execution time, improves the speedup when small chunks of work (followed by a barrier) are parallelized, and also allows programmers to profitably employ barriers to synchronize yet smaller parallel work blocks.

$$speedup_{np} = \frac{W_1}{W_{np} + B_{np}} \tag{8}$$

When synchronizing 18 processes, the effective execution times of the barriers ranged between .12 ms to 1.32 ms across all the experiments; with ranges of .26 ms to 1.32 ms for fixed length work, .15 ms to 1.07 ms for variable length work, and .12 ms to .38 ms for fixed length work with an imbedded critical section. The variance in these times is mostly due to the linear algorithms, whose performance is quite dependent on the type of work that they synchronize. The tree barriers had much more stable execution times across the experiments. For reference, when synchronizing 18 processes, bartre ranged between .38 to .44 ms per iteration across the three experiments, and bbrtre ranged between .26 to .29 ms.

In order to place the barrier execution times into perspective, let us compare their effective depth with the execution speed of the following single precision Fortran vector calculation: C(i) = C(i)+A(i)*B(i). The National Semiconductor NS32032s used in the Flex/32 (Greenhills compiler) require about .038 ms to compute this sum and product for each iteration of the vector index i. This measurement includes Fortran DO loop overhead and index calculations as well as the floating point multiplication and addition. Thus, the range of barrier times, .12 to 1.32 ms, maps into a range of 3 to 35 of these sequential vector element multiply-additions. As an example, the effective depth of bbrtre synchronizing 18 processes, .26 ms, is roughly equivalent to 7 of these vector element multiply-additions.

The primary advantage of the tree barriers is their logarithmic depth. As **np** becomes large, this advantage becomes overwhelming, as demonstrated in Figure 4-2, the timing results for barriers synchronizing fixed length work. While the per stage execution time of the tree barriers is higher than that of the linear barriers, considering Figure 4-2, we see that, by interpolation to larger values of **np**, the tree barriers could all be expected to overtake their linear counterparts when **np** becomes large

enough. For example, using the results obtained on the *Flex/32*, bbrtre would be expected to overtake even bbrlin, the most efficient linear barrier synchronizing fixed length work, for values of **np** near 20. However, if there are critical sections between successive iterations of the barriers, then the self-scheduled linear barriers (barlok, bbrlok) are almost ideally efficient, while the depth of a tree barrier remains proportional to log2(**np**). These tree barriers have the same logarithmic depth as the proposed butterfly barrier.

On the Flex/32, the tree barriers that use only controlled access to synchronizing variables are more efficient than those that use spinlocks. One observation is that the tree barriers using locks tended to show nearly linear depth on the Flex/32, due to the bus contention problem caused by the layers of spinlocks. In fact, the indivisible read-write bus cycles are unnecessary for the tree barriers, and they tie up the shared memory bus(es) longer than necessary. In general, spinlocks tend to require subroutine linkage or possibly inefficient operating system calls, and the spinlocks involve additional computational steps than the set/clear mechanisms. However, one should keep in mind that spinlocks may provide superior performance on machines with special hardware supporting locks.

The issue of whether to use spinlocks or not is a different matter all together for the linear barriers. The linear spinlock barriers allow the processes to arrive in any order, *self-scheduling* the entry into the critical section, and they very effectively exploit any variation in their arrival times. The pre-scheduled linear barriers (using set/clear mechanisms) require a set order of arrival of the processes in order to achieve their best performance. These barriers are also able to exploit variations in process arrival times, but to a lesser degree than their self-scheduled counterparts. Thus, the self-scheduled barriers that use spinlock routines are appealing. However, consider the following trade off. On one hand, the self-scheduled algorithms are better able to exploit significantly skewed process arrival patterns, resulting in better performance for this case, as demonstrated in Figures 4-3 and 4-4. However, due to the critical sections, each stage of the self-scheduled barriers requires more execution time than the corresponding stage of the pre-scheduled barriers. This situation occurs even on the Flex/32 which supports a machine language test and set instruction used to implement the critical sections. So on the other hand, when processes arrive all at once and the effective depth of a linear barrier is the sum of its stages, then the linear pre-scheduled barriers show better performance than the self-scheduled barriers, as shown in Figure 4-2.

All of the barrier algorithms developed in this paper have analogous versions using either symmetric entry and signal phases, or the broadcast exit/polarity idea developed above. On the Flex/32, the broadcast versions show superior performance than their symmetric counterparts. The broadcast exit reduces the exit depth from **np** or log2(**np**) to one, while requiring only minimal computation. If the underlying machine hardware supports true parallel reads of shared data, then the broadcast exit mechanism is almost ideally efficient. If the machine hardware does not support true parallel reads, then the situation where many processes compete to read the exit variable is like a linear critical section, but with a very short time quantum, a single shared memory bus cycle. Given this situation, a very large number of processes, and a machine with multiple, hashed, shared memory modules, where memory references to distinct modules may proceed in parallel, then the symmetric tree barriers (bartre, bartrl) could conceivably yield better performance since they eliminate the competition to read the single exit variable. Care would have to be taken to insure that the synchronizing variables are kept in different memory modules.

5.2. Recommendations

It is an interesting result that the tree barriers show better performance for the case of fixed length work, while the linear self-scheduled barriers show improved performance for variable length work and better performance for fixed length work that contains a critical section. One consequence is that linear barriers are well suited to synchronizing self-scheduled parallel loops, while tree barriers are better suited to synchronizing pre-scheduled homogeneous loops. For finely tuned applications, it may be desirable to tailor the barrier to the work it synchronizes in order to achieve optimal performance. Perhaps in the future, an intelligent compiler may be able to make this decision on a case by case basis. However, in the present day for general applications it would seem easier to decide on a single default barrier in order to insulate the parallel programmer from this type of decision.

Before selecting a default barrier for use on a particular machine architecture, it would be wise to try out several of the algorithms, due to the wide variance and peculiarity of the shared memory multiprocessors currently available. However, if general recommendations can be made, then the barriers should be chosen based on whatever desirable theoretical attributes they possess. For larger values of **np** (**np** > ~8), **bbrtre**, the tree broadcast barrier without locks, is recommended for general applications due to its logarithmic depth and competitive execution times. For the smaller values of **np** (**np** < ~8) and/or applications with many critical sections, **bbrlok**, the self-scheduled linear barrier with broadcast exit is a good choice. This barrier always delivers good performance for small values of **np**, and for larger values of **np** it also performs well when it is able to exploit the run time conditions often associated with critical sections. Both of these barriers, coded in an extended Fortran, are shown in Appendix A.

References

[1] T. S. Axelrod, "Effects of synchronization barriers on multiprocessor performance", *Parallel Computing*, Vol 3, No. 2 (May 1986).

[2] H. F. Jordan, "Parallel computation with the Force", *ICASE Rept. No. 85-45*, NASA Langley Res. Ctr., Hampton, VA (Oct. 1985).

[3] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer," *IEEE Trans. on Computers*, Vol. C-32, No. 2 (Feb. 1983).

[4] *Multicomputing multitasking operating system (MMOS) reference manual*, Flexible Computer Corporation, Dallas, TX, (1986).

Appendices

Appendix A: Fortran code for barlin, barlok, bartre, bartrl, bbrlin, bbrlok, bbrtre, bbrtrl

The following barriers, coded in an extended Fortran, are included in this Appendix. The extended Fortran allows Private and Shared declarations as well as Spinlock and Unlock primitives. In all cases, the Unlock statement denotes a simple unconditional unlock routine.

barlin:	linear (pre-sched),	no locks,	symmetric entry & signal phases
barlok:	linear (self-sched),	locks,	symmetric
bartre:	tree structured,	no locks,	symmetric
bartrl:	tree structured,	locks,	symmetric
bbrlin:	linear (pre-sched),	no locks,	broadcast exit signal
bbrlok:	linear (self-sched),	locks,	broadcast
bbrtre:	tree structured,	no locks,	broadcast
bbrtrl:	tree structured,	locks,	broadcast

These barriers require a single a priori initialization. One process must execute a call to Sh_init *once* before any barriers are executed, in order to initialize the shared variables. Normally the process that forks the other processes can call Sh_init. The argument to Sh_init should be **np**, the number of processes participating in the barrier. Also each process must execute a call to Pr_init *once* in order to initialize its private data structures. The arguments to Pr_init are **id** and **np**, the process id (numbered from one to **np**) and **np**.

These barriers are coded using subroutines calls. A typical expansion of the barriers follows. The process with **id** = 1 will always execute the sequential part.

```
C       Begin Barrier
        CALL Bar_entry
        if (id .eq. 1) then
          .
          .
          [< optional sequential code block >]
          .
C       End Barrier
        CALL Bar_signal
        end if
```

```
**********************************************************************
* barlin:  linear (pre-scheduled), no locks, symmetric structure
**********************************************************************
            Subroutine Bar_entry
             Shared  common /Shbar/ logical LARRAY(20)
             Private common /Prbar/ integer id, np
            if (id. eq. 1) then
               do 10 i=2,np
20                if(LARRAY(i) .eq. .true.) goto 20
10             continue
            else
               LARRAY(id) = .false.
            call Bar_signal
            end if
            return
            end

            Subroutine Bar_signal
             Shared  common /Shbar/ logical LARRAY(20)
             Private common /Prbar/ integer id, np
            if (id. eq. 1) then
               do 10 i=2,np
                  LARRAY(i)=.true.
10             continue
            else
20             if (LARRAY(id) .eq. .false.) goto 20
            end if
            return
            end


**********************************************************************



            Subroutine Pr_init(tid,tnp)
             integer tid,tnp
             Private common /Prbar/ integer id, np
            id = tid
            np = tnp
            return
            end

            Subroutine Sh_init(tnp)
             integer tnp
             Shared  common /Shbar/ logical LARRAY(20)
            do 10 i=2,np
               LARRAY(i)= .true.
10          continue
            return
            end
```

```
*************************************************************************
* barlok:  linear (self-scheduled), locks, symmetric structure
*************************************************************************
            Subroutine Bar_entry
             Shared  common /Shbar/ logical ENTRY, EXIT
             Shared  common /Shbar/ integer COUNTER
             Private common /Prbar/ integer id, np, mycount
            SPINLOCK(ENTRY)
             mycount = COUNTER + 1
             COUNTER = mycount
            if (mycount .lt. np)  UNLOCK(ENTRY)
            if (id .eq. 1)  then
10            if (COUNTER .ne. np)  goto 10
              return
            else
              SPINLOCK(EXIT)
            end if
            mycount = COUNTER - 1
            COUNTER = mycount
            if (mycount .eq. 0)  then
              UNLOCK(ENTRY)
            else
              UNLOCK(EXIT)
            end if
            return
            end

            Subroutine Bar_signal
             Shared  common /Shbar/ logical ENTRY, EXIT
             Shared  common /Shbar/ integer COUNTER
             Private common /Prbar/ integer id, np, mycount
            COUNTER = COUNTER - 1
            UNLOCK(EXIT)
            return
            end

*************************************************************************

            Subroutine Pr_init(tid,tnp)
             integer tid,tnp
             Private common /Prbar/ integer id, np, mycount
            id = tid
            np = tnp
            return
            end

            Subroutine Sh_init(tnp)
             integer tnp
             Shared  common /Shbar/ logical ENTRY, EXIT
             Shared  common /Shbar/ integer COUNTER
            COUNTER = 0
            UNLOCK(ENTRY)
            UNLOCK(EXIT)
            SPINLOCK(EXIT)
            return
            end
```

```
*******************************************************************************
* bartre:  tree structured, no locks, symmetric structure
*******************************************************************************
            Subroutine Bar_entry
              Shared  common /Shbar/ logical LARRAY(20)
              Private common /Prbar/ integer id, np, lim
10          lim=lim/2
20          if (id .le. lim) then
               if ((id+lim) .gt. np) goto 10
30               if (LARRAY(id+lim) .eq. .true.) goto 30
                 goto 10
            end if
            LARRAY(id) = .false.
            if (id .ne. 1) call Bar_signal
            return
            end

            Subroutine Bar_signal
              Shared  common /Shbar/ logical LARRAY(20)
              Private common /Prbar/ integer id, np, lim
            if (id .ne. 1) goto 10
              lim = 1
            goto 30
10          if (LARRAY(id) .eq. .false.) goto 10
20          lim=lim * 2
30          if((id+lim) .le. np) then
               LARRAY(id+lim) = .true.
               goto 20
            end if
            return
            end

*******************************************************************************

            Subroutine Pr_init(tid,tnp)
              integer tid,tnp
              Private common /Prbar/ integer id, np, lim
            id = tid
            np = tnp
C       initialize lim such that:  lim=2**n >= np > 2**(n-1)
            lim = 1
10          if (lim .lt. np) then
               lim = lim * 2
               goto 10
            end if
            return
            end

            Subroutine Sh_init(tnp)
              integer tnp
              Shared  common /Shbar/ logical LARRAY(20)
            do 10 i=2,np
               LARRAY(i)= .true.
10          continue
            return
            end
```

```
*******************************************************************************
* bartrl:  tree structured, locks, symmetric structure
*******************************************************************************
            Subroutine Bar_entry
              Shared  common /Shbar/ logical INARRAY(20), OUTARRAY(20)
              Private common /Prbar/ integer id, np, lim
10          lim=lim/2
20          if (id .le. lim) then
              if ((id+lim) .gt. np) goto 10
30            SPINLOCK(INARRAY(id+lim))
              goto 10
            end if
            UNLOCK(INARRAY(id))
            if (id .ne. 1) CALL Bar_signal
            return
            end

            Subroutine Bar_signal
              Shared  common /Shbar/ logical INARRAY(20), OUTARRAY(20)
              Private common /Prbar/ integer id, np, lim
            if (id .ne. 1) goto 10
            lim = 1
            goto 30
10          SPINLOCK(OUTARRAY(id))
20          lim = lim*2
30          if((id+lim) .le. np) then
              UNLOCK(OUTARRAY(id+lim))
              goto 20
            end if
            return
            end

*******************************************************************************

            Subroutine Pr_init(tid,tnp)
              integer tid,tnp
              Private common /Prbar/ integer id, np, lim
            id = tid
            np = tnp
C       initialize lim such that:  lim=2**n  >=  np  >  2**(n-1)
            lim = 1
10          if (lim .lt. np) then
              lim = lim * 2
              goto 10
            end if
            return
            end

            Subroutine Sh_init(tnp)
              integer tnp
              Shared  common /Shbar/ logical INARRAY(20), OUTARRAY(20)
            do 10 i=1,np
              UNLOCK(INARRAY(i))
              UNLOCK(OUTARRAY(i))
              SPINLOCK(INARRAY(i))
              SPINLOCK(OUTARRAY(i))
10          continue
            return
            end
```

```
**************************************************************************
* bbrlin:  linear (pre-scheduled), no locks, broadcast exit
**************************************************************************
            Subroutine Bar_entry
              Shared  common /Shbar/ logical LARRAY(20)
              Private common /Prbar/ integer id, np, polarity
            if (id .eq. 1) then
               do 10 i = 2,np
20                if (LARRAY(i) .ne. polarity) goto 20
10             continue
            else
               LARRAY(id) = polarity
               polarity =  .not. polarity
30             if (LARRAY(1) .eq. polarity) goto 30
            end if
            return
            end

            Subroutine Bar_signal
              Shared  common /Shbar/ logical LARRAY(20)
              Private common /Prbar/ integer id, np, polarity
            LARRAY(1) = polarity
            polarity =  .not. polarity
            return
            end


**************************************************************************


            Subroutine Pr_init(tid,tnp)
              integer tid,tnp
              Private common /Prbar/ integer id, np, polarity
            id = tid
            np = tnp
            polarity = .true.
            return
            end

            Subroutine Sh_init(tnp)
              integer tnp
              Shared  common /Shbar/ logical LARRAY(20)
            do 10 i=1,np
               LARRAY(i)= .false.
10          continue
            return
            end
```

```
*************************************************************************
* bbrlok:  linear (self-scheduled), locks, broadcast exit
*************************************************************************
            Subroutine Bar_entry
             Shared  common /Shbar/ logical ENTRY, EXIT
             Shared  common /Shbar/ integer COUNTER
             Private common /Prbar/ integer id, np, polarity
            if (id .eq. 1) then
10             if (COUNTER .ne. 0) goto 10
            else
               SPINLOCK(ENTRY)
               COUNTER=COUNTER-1
               UNLOCK(ENTRY)
               polarity = .not. polarity
20             if (EXIT .eq. polarity) goto 20
            end if
            return
            end

            Subroutine Bar_signal
             Shared  common /Shbar/ logical ENTRY, EXIT
             Shared  common /Shbar/ integer COUNTER
             Private common /Prbar/ integer id, np, polarity
            COUNTER = np - 1
            EXIT = polarity
            polarity = .not. polarity
            return
            end


*************************************************************************


            Subroutine Pr_init(tid,tnp)
             integer tid,tnp
             Private common /Prbar/ integer id, np, polarity
            id = tid
            np = tnp
            polarity = .true.
            return
            end

            Subroutine Sh_init(tnp)
             integer tnp
             Shared  common /Shbar/ logical ENTRY, EXIT
            COUNTER = np -1
            EXIT = .false.
            UNLOCK(ENTRY)
            return
            end
```

```
*************************************************************************
* bbrtre:  tree structured, no locks, broadcast exit
*************************************************************************
            Subroutine Bar_entry
             Shared  common /Shbar/ logical LARRAY(20)
             Private common /Prbar/ integer id, np, lim, polarity
             Private integer ilim, isum
            ilim = lim
            goto 20

10          ilim=ilim/2
20          if (id .le. ilim) then
              isum = id + ilim
              if (isum .gt. np) goto 10
30            if (LARRAY(isum) .ne. polarity) goto 30
              goto 10
            end if
            if (id .ne. 1) then
              LARRAY(id) = polarity
              polarity = .not. polarity
40            if (LARRAY(1) .eq. polarity) goto 40
            end if
            return
            end

            Subroutine Bar_signal
             Shared  common /Shbar/ logical LARRAY(20)
             Private common /Prbar/ integer id, np, lim, polarity
            LARRAY(1) = polarity
            polarity =  .not. polarity
            return
            end

*************************************************************************

            Subroutine Pr_init(tid,tnp)
             integer tid,tnp
             Private common /Prbar/ integer id, np, lim, polarity
            id = tid
            np = tnp
            polarity = .true.
C     initialize lim such that:  lim=2**n  <  np  <=  2**(n+1)
            lim  =  1
10          if (lim .lt. np) then
              lim  =  lim * 2
              goto 10
            end if
            lim = lim / 2
            return
            end

            Subroutine Sh_init(tnp)
             integer tnp
             Shared  common /Shbar/ logical LARRAY(20)
            do 20 i = 1,tnp
              LARRAY(i)  =  .false.
20          continue
            return
            end
```

```
*************************************************************************
* bbrtrl:  tree structured, locks, broadcast exit
*************************************************************************
            Subroutine Bar_entry
             Shared  common /Shbar/ logical INARRAY(20)
             Private common /Prbar/ integer id, np, lim, polarity
             Private integer ilim, isum
            ilim=lim
            goto 20
10          ilim=ilim/2
20          if (id .le. ilim) then
              isum = id + ilim
              if (isum .gt. np) goto 10
30            SPINLOCK(INARRAY(isum))
              goto 10
            end if
            if (id .ne. 1) then
              UNLOCK(INARRAY(id))
              polarity = .not. polarity
40            if (INARRAY(1) .eq. polarity) goto 40
            end if
            return
            end

            Subroutine Bar_signal
             Shared  common /Shbar/ logical INARRAY(20)
             Private common /Prbar/ integer id, np, lim, polarity
            INARRAY(1) = polarity
            polarity = .not. polarity
            return
            end

*************************************************************************

            Subroutine Pr_init(tid,tnp)
             integer tid,tnp
             Private common /Prbar/ integer id, np, lim, polarity
            id = tid
            np = tnp
            polarity = .true.
C           initialize lim such that:  lim=2**n  <  np  <=  2**(n+1)
            lim  =  1
10          if (lim .lt. np) then
              lim  =  lim * 2
              goto 10
            end if
            lim = lim / 2
            return
            end

            Subroutine Sh_init(tnp)
             integer tnp
             Shared  common /Shbar/ logical INARRAY(20)
            INARRAY(1) = .false.
            do 10 i=2,np
              UNLOCK(INARRAY(i))
              SPINLOCK(INARRAY(i))
10          continue
            return
            end
```

Appendix B: Compiler issues: optimized out references to shared data?

Consider the following two lines of Fortran code. Statement pairs like this can occur in several of the barriers that have been developed in this paper.

```
        flag = .true.
10      if (flag .eq. .true) goto 10
```

If flag is a shared variable, and another process is expected to set flag to false, then we see that this pair of statements is perfectly reasonable. However, if a conventional high performance optimizing compiler got hold of these two lines, then it might well optimize out the second reference to flag, causing an infinite loop.

What is needed is a new generation of compilers designed for parallel languages. Such compilers would be free to fully optimize references to private variables, storing them in machine registers, etc. But, compilers for parallel languages should, in general, never optimize out references to shared variables in the code that they produce.

Since many present compilers do not meet this requirement, it may be necessary to fool a compiler, so that it will not remove memory references to shared variables. One simple way to do this is to put one or both of the statements in the above example into a subroutine. For a language such as Fortran, assuming parameters are passed by address (and not with copy-restore), this "quick fix" is sufficient to insure that all required references to shared variables actually occur. This is the approach that has been adopted for the algorithms coded in Appendix A. A second alternative is to code in assembly language. Still another alternative is to thoroughly understand the compiler to be used, before programming in a compiled high level language.

BIBLIOGRAPHIC DATA SHEET	1. Report No. ECE Tech. Rept. 87-1-2	2.	3. Recipient's Accession No.

4. Title and Subtitle

Comparing Barrier Algorithms

5. Report Date: June 1987

6.

7. Author(s): Norbert S. Arenstorf and Harry F. Jordan

8. Performing Organization Rept. No. CSDG 87-3

9. Performing Organization Name and Address

Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425

10. Project/Task/Work Unit No.

11. Contract/Grant No. NAG-1-6[illegible]

12. Sponsoring Organization Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

13. Type of Report & Period Covered: Interim

14.

15. Supplementary Notes

16. Abstracts

A barrier is a method for synchronizing a large number of concurrent computer processes. After considering some basic synchronization mechanisms, a collection of barrier algorithms with either linear or logarithmic depth will be presented. A graphical model is described that profiles the execution of the barriers and other parallel programming constructs. This model shows how the interaction between the barrier algorithms and the work that they synchronize can impact their performance. One result is that logarithmic tree structured barriers show good performance synchronizing fixed length work; while linear self-scheduled barriers show better performance when synchronizing fixed length work with an imbedded critical section. The linear barriers are better able to exploit the process skew associated with critical sections. Timing experiments, performed on an eighteen processor *Flex/32* shared memory multiprocessor, that support these conclusions are detailed.

17. Key Words and Document Analysis. 17a. Descriptors

multiprocessor
shared memory
synchronization
performance

17b. Identifiers/Open-Ended Terms

barrier
Flex/32

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 38
	20. Security Class (This Page) UNCLASSIFIED	22. Price

END

DATE

FILM

DTIC

7—85